1.4: Maths libraries

Tony Debling, SGS-Thomson Microelectronics Ltd, 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, England. June 7, 1995.
Converted to HTML via RTFtoHTML, PERL5 and by hand by Dave Beckett <*D.J.Beckett@ukc.ac.uk*>.

WARNING: This has been converted from RTF format and the formulas and expressions have not been checked with the printed version. If you find any errors in this file, please send them to me, and I will update it.

Dave Beckett <*D.J.Beckett@ukc.ac.uk*>

Elementary maths and trigonometric functions are provided in three libraries, as follows:

```
Library         Description
snglmath.lib    Single length library
dblmath.lib     Double length library
tbmaths.lib     TB optimized library
```

The single and double length libraries contain the same set of maths functions in single and double length forms. The double length forms all begin with the letter `D'. All function names are in upper case.

The TB optimized library is a combined single and double length library containing functions for the T4 series (T400, T414, T425, and T426). The functions have been optimized for speed. The standard single or double length libraries *can* be used on T4 processors but optimum performance will be achieved by using the TB optimized library. The accuracy of the T400/T414/T425/T426 optimized functions is similar to that of the standard single length functions but results returned may not be identical because different algorithms are used. If the optimized library is used in code compiled for any processor except a T400, T414, T425, or T426, the compiler reports an error.

To obtain the best possible speed performance with the occam maths functions use the following strategy:

- For networks consisting of only T4 series transputers, use the **`tbmaths.lib`** library.
- For networks consisting of only T8 series transputers, use the **`snglmath.lib`** and **`dblmath.lib`** libraries.
- For networks consisting of a mix of T4 series and T8 series transputers use:
  - **`tbmaths.lib`** on the T4 series and **`snglmath.lib`** or **`dblmath.lib`** on the T8 series when a consistent level of accuracy is not required;
  - if accuracy must be the same in the T8 and T4 processes then use the **`snglmath.lib`** and **`dblmath.lib`** libraries.

Constants for the maths libraries are provided in the include file **`mathvals.inc`**.

The elementary function library is also described in appendix N of the *occam 2 Reference manual*.

1.4.1: Introduction and terminology

This, and the following subsections, contain some notes on the presentation of the elementary function libraries described in section 1.4.2, and the TB version described in section 1.4.3.

These function subroutines have been written to be compatible with the ANSI standard for binary floating-point arithmetic (ANSI IEEE std 754-1985), as implemented in occam. They are based on the algorithms in:

> Cody, W. J., and Waite, W. M. [1980]. *Software Manual for the Elementary Functions.* Prentice-Hall, New Jersey.

The only exceptions are the pseudo-random number generators, which are based on algorithms in:

> Knuth, D. E. [1981]. *The Art of Computer Programming, 2nd. edition, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Mass.

Inputs

All the functions in the library (except `RAN` and `DRAN`) are called with one or two parameters which are binary floating-point numbers in one of the IEEE standard formats, either `single-length' (32 bits) or `double-length' (64 bits). The parameter(s) and the function result are of the same type.

NaNs and Infs

The functions will accept any value, as specified by the standard, including special values representing **NaN**s (`Not a Number') and **Inf**s (`Infinity'). **NaN**s are copied to the result, whilst **Inf**s may or may not be in the domain. The domain is the set of arguments for which the result is a normal (or denormalized) floating-point number.

Outputs
Exceptions

Arguments outside the domain (apart from **NaNs** which are simply copied through) give rise to *exceptional results*, which may be **NaN**, +**Inf**, or -**Inf**. **Inf**s mean that the result is mathematically well-defined but too large to be represented in the floating-point format.

Error conditions are reported by means of three distinct **NaN**s:

undefined.NaN

This means that the function is mathematically undefined for this argument, for example the logarithm of a negative number.

unstable.NaN

This means that a small change in the argument would cause a large change in the value of the function, so *any* error in the input will render the output meaningless.

inexact.NaN

This means that although the mathematical function is well-defined, its value is in range, and it is stable with respect to input errors at this argument, the limitations of word-length (and reasonable cost of the algorithm) make it impossible to compute the correct value.

The implementations will return the following values for these Not-a-Numbers:

Error	Single length value	Double length value
undefined.NaN	#7F800010	#7FF00002 00000000
unstable.NaN	#7F800008	#7FF00001 00000000
inexact.NaN	#7F800004	#7FF00000 80000000

Accuracy
Range Reduction

Since it is impractical to use rational approximations (i.e. quotients of polynomials) which are accurate over large domains, nearly all the subroutines use mathematical identities to relate the function value to one computed from a smaller argument, taken from the `primary domain', which is small enough for such an approximation to be used. This process is called `range reduction' and is performed for all arguments except those which already lie in the primary domain.

For most of the functions the quoted error is for arguments in the primary domain, which represents the basic accuracy of the approximation. For some functions the process of range reduction results in a higher accuracy for arguments outside the primary domain, and for others it does the reverse. Refer to the notes on each function for more details.

Generated Error

If the true value of the function is large the difference between it and the computed value (the `absolute error') is likely to be large also because of the limited accuracy of floating-point numbers. Conversely if the true value is small, even a small absolute error represents a large proportional change. For this reason the error relative to the true value is usually a better measure of the accuracy of a floating-point function, except when the output range is strictly bounded.

If f is the mathematical function and F the subroutine approximation, then the relative error at the floating-point number X (provided $f(X)$ is not zero) is:

Obviously the relative error may become very large near a zero of $f(X)$. If the zero is at an irrational argument (which cannot be represented as a floating-point value), the absolute error is a better measure of the accuracy of the function near the zero.

As it is impractical to find the relative error for every possible argument, statistical measures of the overall error must be used. If the relative error is sampled at a number of points Xn (n = 1 to N), then useful statistics are the *maximum relative error* and the *root-mean-square relative error*:

Corresponding statistics can be formed for the absolute error also, and are called *MAE* and *RMSAE* respectively.

The *MRE* generally occurs near a zero of the function, especially if the true zero is irrational, or near a singularity where the result is large, since the `granularity' of the floating-point numbers then becomes significant.

A useful unit of relative error is the relative magnitude of the least significant bit in the floating-point fraction, which is called one `unit in the last place' (ulp), (i.e. the smallest e such that 1+e 01). Its magnitude depends on the floating-point format: for single-length it is 2^(-23) = 1.19*10^(-7), and for double-length it is 2^(-52)= 2.22*10^(-16).

Propagated Error

Because of the limited accuracy of floating-point numbers the result of any calculation usually differs from the exact value. In effect, a small error has been added to the exact result, and any subsequent calculations will inevitably involve this error term. Thus it is important to determine how each function responds to errors in its argument. Provided the error is not too large, it is sufficient just to consider the first derivative of the function (written *f'*).

If the relative error in the argument *X* is *d* (typically a few ulp), then the absolute error (*E*) and relative error (*e*) in *f*(*X*) are:

This defines the absolute and relative error magnification factors *A* and *R*. When both are large the function is unstable, i.e. even a small error in the argument, such as would be produced by evaluating a floating-point expression, will cause a large error in the value of the function. The functions return an **unstable.NaN** in such cases which are simple to detect.

The functional forms of both *A* and *R* are given in the specification of each function.

Test Procedures

For each function, the generated error was checked at a large number of arguments (typically 100,000) drawn at random from the appropriate domain. First the double-length functions were tested against a `quadruple-length' implementation (constructed for accuracy rather than speed), and then the single-length functions were tested against the double-length versions.

In both cases the higher-precision implementation was used to approximate the mathematical function (called *f* above) in the computation of the error, which was evaluated in the higher precision to avoid rounding errors. Error statistics were produced according to the formulae above.

Symmetry

The subroutines were designed to reflect the mathematical properties of the functions as much as possible. For all the functions which are even, the sign is removed from the input at the beginning of the computation so that the sign-symmetry of the function is always preserved. For odd functions, either the sign is removed at the start and then the appropriate sign set at the end of the computation, or else the sign is simply propagated through an odd degree polynomial. In many cases other symmetries are used in the range-reduction, with the result that they will be satisfied automatically.

The Function Specifications

Names and Parameters

All single length functions except **RAN** return a single result of type **REAL32**, and all except **RAN**, **POWER** and **ATAN2** have one parameter, a **VAL REAL32**.

POWER and **ATAN2** have two parameters which are **VAL REAL32**s for the two arguments of each function.

RAN returns two results, of types **REAL32** and **INT32**, and has one parameter which is a **VAL INT32**.

In each case the double-length version of *name* is called **D***name*, returns a **REAL64** (except **DRAN**, which returns **REAL64**, **INT64**), and has parameters of type **VAL REAL64** (**VAL INT64** for **DRAN**).

Terms used in the Specifications

- **A and R** Multiplying factors relating the absolute and relative errors in the output to the relative error in the argument.
- **Exceptions** Outputs for invalid inputs (i.e. those outside the *domain*), other than **NaN** (**NaN**s are copied directly to the output and are not listed as exceptions). These are all **Inf**s or **NaN**s.
- **Generated Error** The difference between the true and computed values of the function, when the argument is error-free. This is measured statistically and displayed for one or two ranges of arguments, the first of which is usually the *primary domain* (see below). The second range, if present, is chosen to illustrate the typical behavior of the function.
- **Domain** The range of valid inputs, i.e. those for which the output is a normal or denormal floating-point number.
- **MAE and RMSAE** The Maximum Absolute Error and Root-Mean-Square absolute error taken over a number of arguments drawn at random from the indicated range.
- **MRE and RMSRE** The Maximum Relative Error and Root-Mean-Square relative error taken over a number of arguments drawn at random from the indicated range.
- **Range** The range of outputs produced by all arguments in the *Domain*. The given endpoints are not exceeded.
- **Primary Domain** The range of arguments for which the result is computed using only a single rational approximation to the function. There is no argument reduction in this range.
- **Propagated Error** The absolute and relative error in the function value, given a small relative error in the argument.
- **ulp** The unit of relative error is the `unit in the last place' (ulp). This is the relative magnitude of the least significant bit of the floating-point fraction (i.e. the smallest e such that 1+e 0 1).

N.B. this depends on the floating-point format.

For the standard single-length format it is 2^(-23) = 1.19*10^(-7).

For the double-length format it is 2^(-52) = 2.22*10^(-16).

This is also used as a measure of absolute error, since such errors can be considered `relative' to unity.

Specification of Ranges

Ranges are given as intervals, using the convention that a square bracket `[' or `]' means that the adjacent endpoint is included in the range, whilst a round bracket `(' or `)' means that it is excluded. Endpoints are given to a few significant figures only.

Where the range depends on the floating-point format, single-length is indicated with an S and double-length with a D.

For functions with two arguments the complete range of both arguments is given. This means that for each number in one range, there is at least one (though sometimes only one) number in the other range such that the pair of arguments is valid. Both ranges are shown, linked by an `x'.

Abbreviations

In the specifications, *XMAX* is the largest representable floating-point number: in single-length it is approximately 3.4*10^(38), and in double-length it is approximately 1.8*10^(308).

Pi means the closest floating-point representation of the transcendental number p, ln(2) the closest representation of log *e*(2), and so on.

In describing the algorithms, `*X*' is used generically to designate the argument, and `result' (or RESULT, in the style of occam functions) to designate the output.

1.4.2: Single and double length elementary function libraries

The versions of the libraries described by this section have been written using only floating-point arithmetic and pre-defined functions supported in occam. Thus they can be compiled for any processor with a full implementation of occam, and give identical results.

These two libraries will be efficient on processors with fast floating-point arithmetic and good support for the floating-point predefined functions such as **MULBY2** and **ARGUMENT.REDUCE**. For 32-bit processors without special hardware for floating-point calculations the alternative optimized library described in section 1.4.3 using fixed-point arithmetic will be faster, but will not give identical results.

A special version has been produced for 16-bit transputers, which avoids the use of any double-precision arithmetic in the single precision functions. This is distinguished in the notes by the annotation `T2 special'; notes relating to the version for T8 and TB are denoted by `standard'.

Single and double length maths functions are listed below. Descriptions of the functions can be found in succeeding sections.

To use the single length library a program header must include the line

```
#USE "snglmath.lib"
```

To use the double length library a program header must include the line

```
#USE "dblmath.lib"
```

Result(s)	Function	Parameter specifiers
REAL32	ALOG	VAL REAL32 X
REAL32	ALOG10	VAL REAL32 X
REAL32	EXP	VAL REAL32 X
REAL32	POWER	VAL REAL32 X, VAL REAL32 Y
REAL32	SIN	VAL REAL32 X
REAL32	COS	VAL REAL32 X
REAL32	TAN	VAL REAL32 X
REAL32	ASIN	VAL REAL32 X
REAL32	ACOS	VAL REAL32 X
REAL32	ATAN	VAL REAL32 X
REAL32	ATAN2	VAL REAL32 X, VAL REAL32 Y
REAL32	SINH	VAL REAL32 X
REAL32	COSH	VAL REAL32 X
REAL32	TANH	VAL REAL32 X
REAL32,INT32	RAN	VAL INT32 X
REAL64	DALOG	VAL REAL64 X
REAL64	DALOG10	VAL REAL64 X
REAL64	DEXP	VAL REAL64 X
REAL64	DPOWER	VAL REAL64 X, VAL REAL64 Y
REAL64	DSIN	VAL REAL64 X
REAL64	DCOS	VAL REAL64 X
REAL64	DTAN	VAL REAL64 X
REAL64	DASIN	VAL REAL64 X
REAL64	DACOS	VAL REAL64 X
REAL64	DATAN	VAL REAL64 X
REAL64	DATAN2	VAL REAL64 X, VAL REAL64 Y
REAL64	DSINH	VAL REAL64 X
REAL64	DCOSH	VAL REAL64 X
REAL64	DTANH	VAL REAL64 X
REAL64,INT64	DRAN	VAL INT64 X

Function definitions

ALOG
DALOG

```
REAL32 FUNCTION ALOG (VAL REAL32 X)
REAL64 FUNCTION DALOG (VAL REAL64 X)
```

Compute loge(*X*).

```
Domain:              (0, XMAX]
Range:               [MinLog, MaxLog] [MinLog, MaxLog] (See note 2)
Primary Domain:      [p2/2, p2) = [0.7071, 1.4142)
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A 5 1, *R* +1/loge(*X*)

Generated Error

```
Primary Domain Error:          MRE        RMSRE
Single Length(Standard):       1.7 ulp    0.43 ulp
Single Length(T2 special):     1.6 ulp    0.42 ulp
Double Length:                 1.4 ulp    0.38 ulp
```

The Algorithm

1. Split *X* into its exponent *N* and fraction *F*.
2. Find *LnF*, the natural log of *F*, with a floating-point rational approximation.
3. Compute ln(2) * *N* with extended precision and add it to *LnF* to get the result.

Notes

1) The term ln(2) * *N* is much easier to compute (and more accurate) than *LnF*, and it is larger provided *N* is not 0 (i.e. for arguments outside the primary domain). Thus the accuracy of the result improves as the modulus of log(*X*) increases.

2) The minimum value that can be produced, *MinLog*, is the logarithm of the smallest denormalized floating-point number. For single length *MinLog* is -103.28, and for double length it is -744.4. The maximum value *MaxLog* is the logarithm of *XMAX*. For single-length it is 88.72, and for double-length it is 709.78.

3) Since **Inf** is used to represent *all* values greater than *XMAX* its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

ALOG10
DALOG10

```
REAL32 FUNCTION ALOG10 (VAL REAL32 X)
REAL64 FUNCTION DALOG10 (VAL REAL64 X)
```

Compute log10(*X*).

```
Domain:              (0, XMAX]
Range:               [MinL10, MaxL10] (See note 2)
Primary Domain:      [p2/2, p2) = [0.7071, 1.4142)
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A == log10(*e*), *R* = log10(*e*)/log*e*(*X*)

Generated Error

```
Primary Domain Error:           MRE        RMSRE
Single Length (Standard):       1.70 ulp   0.45 ulp
Single Length (T2 special):     1.71 ulp   0.46 ulp
Double Length:                  1.84 ulp   0.45 ulp
```

The Algorithm

1. Set *temp*:= **ALOG(X)**.
2. If *temp* is a **NaN**, copy it to the output, otherwise set result = log(*e*) * *temp*

Notes

1) See note 1 for **ALOG**.

2) The minimum value that can be produced, *MinL*10, is the base 10 logarithm of the smallest denormalized floating-point number. For single length *MinL*10 is -44.85, and for double length it is -323.3. The maximum value *MaxL*10 is the base 10 logarithm of *XMAX*. For single length *MaxL*10 is 38.53, and for double-length it is 308.26.

3) Since **Inf** is used to represent *all* values greater than *XMAX* its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

EXP
DEXP

```
REAL32 FUNCTION EXP (VAL REAL32 X)
REAL64 FUNCTION DEXP (VAL REAL64 X)
```

Compute *e^(X)*.

```
Domain:              [-Inf, MaxLog) = [-Inf, 88.72)S, [-Inf, 709.78)D
Range:               [0, Inf) (See note 4)
Primary Domain:      [-Ln2/2, Ln2/2) = [-0.3466, 0.3466)
```

Exceptions

All arguments outside the domain generate an **Inf**.

Propagated error

A = Xe^(X), R = X

Generated error

```
Primary Domain Error:           MRE        RMSRE
Single Length(Standard):        0.99 ulp   0.25 ulp
Single Length(T2 special):      1.0 ulp    0.25 ulp
Double Length:                  1.4 ulp    0.25 ulp
```

The Algorithm

1. Set *N* = integer part of *X*/ln(2).
2. Compute the remainder of *X* by ln(2), using extended precision arithmetic.
3. Compute the exponential of the remainder with a floating-point rational approximation.
4. Increase the exponent of the result by *N*. If *N* is sufficiently negative the result must be denormalized.

Notes

1) *MaxLog* is log *e*(*XMAX*).

2) For sufficiently negative arguments (below -87.34 for single-length and below -708.4 for double-length) the output is denormalized, and so the floating-point number contains progressively fewer significant digits, which degrades the accuracy. In such cases the error can theoretically be a factor of two.

3) Although the true exponential function is never zero, for large negative arguments the true result becomes too small to be represented as a floating-point number, and **EXP** underflows to zero. This occurs for arguments below -103.9 for single-length, and below -745.2 for double-length.

4) The propagated error is considerably magnified for large positive arguments, but diminished for large negative arguments.

POWER
DPOWER

```
REAL32 FUNCTION POWER (VAL REAL32 X, Y)
REAL64 FUNCTION DPOWER (VAL REAL64 X, Y)
```

Compute *X^(Y)*.

```
Domain:              [0, Inf] x [-Inf, Inf]
Range:               (-Inf, Inf)
Primary Domain:      See note 3.
```

Exceptions

If the first argument is outside its domain, **undefined.NaN** is returned. If the true value of *X^(Y)* exceeds *XMAX*, **Inf** is returned. In certain other cases other **NaN**s are produced: See note 2.

Propagated Error

A = *YX^(Y)*(1 [[formula]] log *e*(*X*)), *R* = *Y*(1 [[formula]] log *e*(*X*)) (See note 4)

Generated error

```
Example Range Error:                    MRE      RMSRE  (See note 3)
Single Length(Standard):            1.0 ulp   0.25 ulp
Single Length(T2 special):         63.1 ulp   13.9 ulp
Double Length:                     21.1 ulp    2.4 ulp
```

The Algorithm

Deal with special cases: either argument = 1, 0, +**Inf** or -**Inf** (see note 2). Otherwise:

(a) For the standard single precision:

1. Compute *L* = log *e*(*X*) in double precision, where *X* is the first argument.
2. Compute *W* = *Y* x *L* in double precision, where *Y* is the second argument.
3. Compute *RESULT* = *e^(W)* in single precision.

(b) For double precision, and the single precision special version:

1. Compute *L* = log2(*X*) in extended precision, where *X* is the first argument.
2. Compute *W* = *Y* x *L* in extended precision, where *Y* is the second argument.
3. Compute *RESULT* = 2^(*W*) in extended precision.

Notes

1) This subroutine implements the mathematical function *x^(y)* to a much greater accuracy than can be attained using the **ALOG** and **EXP** functions, by performing each step in higher precision. The single-precision version is more efficient than using **DALOG** and **EXP** because redundant tests are omitted.

2) Results for special cases are as follows:

```
First Input (X)        Second Input (Y)            Result
      t0                     ANY                undefined.NaN
       0                      v0                undefined.NaN
       0                 0 <Y<=XMAX                  0
       0                     Inf                unstable.NaN
   0 t X t1                  Inf                    0
   0 t X t1                 -Inf                   Inf
       1              -XMAX<=Y<=XMAX                1
       1                    " Inf                unstable.NaN
 1 t X v XMAX               Inf                    Inf
 1 t X v XMAX              -Inf                     0
      Inf               1 v Y v Inf                Inf
      Inf              -Inf vY v -1                 0
      Inf                -1 t Y t1              undefined.NaN
   otherwise                 0                     1
   otherwise                 1                     X
```

3) Performing all the calculations in extended precision makes the double-precision algorithm very complex in detail, and having two arguments makes a primary domain difficult to specify. As an indication of accuracy, the functions were evaluated at 100,000 points logarithmically distributed over (0.1, 10.0), with the exponent linearly distributed over (-35.0, 35.0) (single-length), and (-300.0, 300.0) (double-length), producing the errors given above. The errors are much smaller if the exponent range is reduced.

4) The error amplification factors are calculated on the assumption that the relative error in *Y* is [[formula]] that in *X*, otherwise there would be separate factors for both *X* and *Y*. It can be seen that the propagated error will be greatly amplified whenever log*e*(*X*) or *Y* is large.

SIN
DSIN

```
REAL32 FUNCTION SIN (VAL REAL32 X)
REAL64 FUNCTION DSIN (VAL REAL64 X)
```

Compute sine(*X*) (where *X* is in radians).

```
Domain: [-Smax, Smax]      = [-205887.4, 205887.4]S (Standard),
=
                           [-4.2*106, 4.2*106]S (T2 special)
=
                           [-4.29*109, 4.29*109]D
Range:                     [-1.0, 1.0]
Primary Domain:            [-Pi/2, Pi/2]= [-1.57, 1.57]
```

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except [[formula]] **Inf**, which generates an **undefined.NaN**.

Propagated Error

$A = X \cos(X)$, $R = X \cot(X)$

Generated error (See note 1)

```
                                Primary Domain            [0, 2Pi]
                              MRE         RMSRE       MAE         RMSAE
Single Length(Standard):      0.94 ulp    0.23 ulp    0.96 ulp    0.19 ulp
Single Length(T2 special):    0.92 ulp    0.23 ulp    0.94 ulp    0.19 ulp
Double Length:                0.90 ulp    0.22 ulp    0.91 ulp    0.18 ulp
```

The Algorithm

1. Set *N* = integer part of *X* *Pi*.
2. Compute the remainder of *X* by *Pi*, using extended precision arithmetic (double precision in the standard version).
3. Compute the sine of the remainder using a floating-point polynomial.
4. Adjust the sign of the result according to the sign of the argument and the evenness of *N*.

Notes

1) For arguments outside the primary domain the accuracy of the result depends crucially on step 2. The extra precision of step 2 is lost if *N* becomes too large, and the cut-off *Smax* is chosen to prevent this. In any case for large arguments the `granularity' of floating-point numbers becomes a significant factor. For arguments larger than *Smax* a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

2) The propagated error has a complex behavior. The propagated relative error becomes large near each zero of the function (outside the primary range), but the

propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

3) Since only the remainder of *X* by *Pi* is used in step 3, the symmetry *sin*(*x*+ *n*p) = [[formula]] *sin* (*x*) is preserved, although there is a complication due to differing precision representations of p.

4) The output range is not exceeded. Thus the output of **SIN** is always a valid argument for **ASIN**.

COS
DCOS

```
REAL32 FUNCTION COS (VAL REAL32 X)
REAL64 FUNCTION DCOS (VAL REAL64 X)
```

Compute cosine(*X*) (where *X* is in radians).

```
Domain: [-Cmax, Cmax]      = [-205887.4, 205887.4]S (Standard),
=
                           [-12868.0, 12868.0]S (T2 special)
=
                           [-2.1*108, 2.1*108]D
Range:                     [-1.0, 1.0]
Primary Domain:            See note 1.
```

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except [[formula]]**Inf**, which generates an **undefined.NaN**.

Propagated Error

A = -*X sin* (*X*), *R* = -*X tan* (*X*) (See note 4)

Generated error

	[0, Pi/4)		[0, 2Pi]	
	MRE	RMSRE	MAE	RMSAE
Single Length(Standard):	0.93 ulp	0.25 ulp	0.88 ulp	0.18 ulp
Single Length(T2 special):	1.1 ulp	0.3 ulp	0.94 ulp	0.19 ulp
Double Length:	1.0 ulp	0.28 ulp	0.90 ulp	0.19 ulp

The Algorithm

1. Set *N* = integer part of (*X*)*Pi* 2) *Pi* and compute the remainder of (*X*)*Pi* 2) by *Pi*, using extended precision arithmetic (double precision in the standard version).
2. Compute the sine of the remainder using a floating-point polynomial.
3. Adjust the sign of the result according to the evenness of *N*.

Notes

1) Inspection of the algorithm shows that argument reduction always occurs, thus there is no `primary domain' for cos. So for all arguments the accuracy of the result depends

crucially on step 2. The standard single-precision version performs the argument reduction in double-precision, so there is effectively no loss of accuracy at this step. For the T2 special version and the double-precision version there are effectively *K* extra bits in the representation of p(*K*=8 for the former and 12 for the latter). If the argument agrees with an odd integer multiple of p/2 to more than *k* bits there is a loss of significant bits from the computed remainder equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The difference between cos evaluated at successive floating-point numbers is given approximately by the absolute error amplification factor, A. For arguments larger than *Cmax* this difference may be more than half the significant bits of the result, and so the result is considered to be essentially indeterminate and an **inexact.NaN** is returned. The extra precision of step 2 in the double-precision and T2 special versions is lost if *N* becomes too large, and the cut-off at *Cmax* prevents this also.

3) For small arguments the errors are not evenly distributed. As the argument becomes smaller there is an increasing bias towards negative errors (which is to be expected from the form of the Taylor series). For the single-length version and *X* in [-0.1, 0.1], 62% of the errors are negative, whilst for *X* in [-0.01, 0.01], 70% of them are.

4) The propagated error has a complex behavior. The propagated relative error becomes large near each zero of the function, but the propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

5) Since only the remainder of (|*X*|+*Pi*/2) by *Pi* is used in step 3, the symmetry cos(*x*+*n*p) = [[formula]] cos(*x*) is preserved. Moreover, since the same rational approximation is used as in **SIN**, the relation cos(*x*) = sin(*x*+p/2) is also preserved. However, in each case there is a complication due to the different precision representations of p.

6) The output range is not exceeded. Thus the output of cos is always a valid argument for **ACOS**.

TAN
DTAN

```
REAL32 FUNCTION TAN (VAL REAL32 X)
REAL64 FUNCTION DTAN (VAL REAL64 X)
```

Compute tan(*X*) (where *X* is in radians).

```
Domain: [-Tmax, Tmax]      = [-102943.7, 102943.7]S(Standard),
=
                           [-2.1*106, 2.1*106]S(T2 special),
=
                           [-2.1*109, 2.1*109]D
Range:                     (-Inf, Inf)
Primary Domain:            [-Pi/4, Pi/4]= [-0.785, 0.785]
```

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except [[formula]] **Inf**, which generate an **undefined.NaN**. Odd integer multiples of p/2 may produce **unstable.NaN**.

Propagated Error

$A = X(1+ \tan^{(2)}(X))$, $R = X(1+ \tan^{(2)}(X))/\tan(X)$ (See note 3)

Generated error

```
Primary Domain Error:                MRE      RMSRE  (See note 3)
Single Length(Standard):         1.44 ulp   0.39 ulp
Single Length(T2 special):       1.37 ulp   0.39 ulp
Double Length:                   1.27 ulp   0.35 ulp
```

The Algorithm

1. Set *N* = integer part of *X*/(*Pi*/2), and compute the remainder of *X* by *Pi*/2, using extended precision arithmetic.
2. Compute two floating-point rational functions of the remainder, *XNum* and *XDen*.
3. If *N* is odd, set *RESULT* = - *XDen*/*XNum*, otherwise set *RESULT* = *XNum*/*XDen*.

Notes

1) *R* is large whenever *X* is near to an integer multiple of p/2, and so tan is very sensitive to small errors near its zeros and singularities. Thus for arguments outside the primary domain the accuracy of the result depends crucially on step 2, so this is performed with very high precision, using double precision *Pi*/2 for the standard single-precision function and two double-precision floating-point numbers for the representation of p /2 for the double-precision function. The T2 special version uses two single-precision floating-point numbers. The extra precision is lost if *N* becomes too large, and the cut-off *Tmax* is chosen to prevent this.

2) The difference between **TAN** evaluated at successive floating-point numbers is given approximately by the absolute error amplification factor, *A*. For arguments larger than *Smax* this difference could be more than half the significant bits of the result, and so the result is considered to be essentially indeterminate and an **inexact.NaN** is returned.

3) Tan is quite badly behaved with respect to errors in the argument. Near its zeros outside the primary domain the relative error is greatly magnified, though the absolute error is only proportional to the size of the argument. In effect, the error is seriously amplified in an interval about each irrational zero, whose width increases roughly in proportion to the size of the argument. Near its singularities both absolute and relative errors become large, so any large output from this function is liable to be seriously contaminated with error, and the larger the argument, the smaller the maximum output which can be trusted. If step 3 of the algorithm requires division by zero, an **unstable.NaN** is produced instead.

4) Since only the remainder of *X* by *Pi*/2 is used in step 3, the symmetry tan(*x*+ *n*p) = tan(*x*) is preserved, although there is a complication due to the differing precision representations of p. Moreover, by step 3 the symmetry tan(*x*) = 1/tan(p/2 - *x*) is also preserved.

ASIN
DASIN

```
REAL32 FUNCTION ASIN (VAL REAL32 X)
REAL64 FUNCTION DASIN (VAL REAL64 X)
```

Compute sine^(-1)(*X*) (in radians).

```
Domain:              [-1.0, 1.0]
Range:               [-Pi/2, Pi/2]
Primary Domain:      [-0.5, 0.5]
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A = *X*/p1 - *X*^(2), *R* = *X*/(sin^(-1)(*X*) p1 - *X*^(2))

Generated Error

```
                                Primary Domain            [-1.0, 1.0]
                              MRE         RMSRE       MAE         RMSAE


Single Length:                0.58 ulp    0.21 ulp    1.35 ulp    0.33 ulp
Double Length:                0.59 ulp    0.21 ulp    1.26 ulp    0.27 ulp
```

The Algorithm

1. If |*X*| > 0.5, set *Xwork*:= SQRT(1 - *X*)2). Compute *Rwork* = arcsine(-2 * *Xwork*) with a floating-point rational approximation, and set the result = *Rwork* + *Pi*/2.
2. Otherwise compute the result directly using the rational approximation.
3. In either case set the sign of the result according to the sign of the argument.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error: however since both domain and range are bounded the *absolute* error in the result cannot be large.

2) By step 1, the identity sin^(-1)(*x*) = p/2 - 2 sin^(-1)(p(1-*x*)/2)) is preserved.

ACOS
DACOS

```
REAL32 FUNCTION ACOS (VAL REAL32 X)
REAL64 FUNCTION DACOS (VAL REAL64 X)
```

Compute cosine^(-1)(*X*) (in radians).

```
Domain:              [-1.0, 1.0]
Range:               [0, Pi]
Primary Domain:      [-0.5, 0.5]
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A = -*X*/p1 - *X*^(2), *R* = -*X*/(sin^(-1)(*X*) p1 - *X*^(2))

Generated Error

```
                              Primary Domain            [-1.0, 1.0]
                            MRE         RMSRE       MAE         RMSAE
Single Length:              1.06 ulp    0.38 ulp    2.37 ulp    0.61 ulp
Double Length:              0.96 ulp    0.32 ulp    2.25 ulp    0.53 ulp
```

The Algorithm

1. If |*X*| > 0.5, set *Xwork*:= **SQRT**((1 - *X*) 2). Compute *Rwork* = arcsine(2 * *Xwork*) with a floating-point rational approximation. If the argument was positive, this is the result, otherwise set the result = *Pi* - *Rwork*.
2. Otherwise compute *Rwork* directly using the rational approximation. If the argument was positive, set result = *Pi*/2 - *Rwork*, otherwise result = *Pi*/2 + *Rwork*.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error, although this interval is larger near 1 than near -1, since the function goes to zero with an infinite derivative there. However since both the domain and range are bounded the *absolute* error in the result cannot be large.

2) Since the rational approximation is the same as that in **ASIN**, the relation cos^(-1)(*x*) = p/2 - sin^(-1)(*x*) is preserved.

ATAN
DATAN

```
REAL32 FUNCTION ATAN (VAL REAL32 X)
REAL64 FUNCTION DATAN (VAL REAL64 X)
```

Compute tan^(-1)(*X*) (in radians).

```
Domain:             [-Inf, Inf]
Range:              [-Pi/2, Pi/2]
Primary Domain:     [-z, z],   z = 2 - p3 = 0.2679
```

Exceptions

None.

Propagated Error

A = *X*/(1 + *X*^(2)), *R* = *X*/(tan^(-1)(*X*)(1 + *X*^(2)))

Generated Error

```
Primary Domain Error:     MRE         RMSRE
Single Length:            0.56 ulp    0.21 ulp
Double Length:            0.52 ulp    0.21 ulp
```

The Algorithm

1. If |*X*| > 1.0, set *Xwork* = 1/|*X*| , otherwise *Xwork* = |*X*|.
2. If *Xwork* > 2-p3, set *F* = (*Xwork**p3 -1)/(*Xwork* +p3), otherwise *F* = *Xwork*.

3. Compute *Rwork* = arctan(*F*) with a floating-point rational approximation.
4. If *Xwork* was reduced in (2), set *R* = *Pi*/6 + *Rwork*, otherwise *R* = *Rwork*.
5. If *X* was reduced in (1), set *RESULT* = *Pi*/2 - *R*, otherwise *RESULT* = *R*.
6. Set the sign of the *RESULT* according to the sign of the argument.

Notes

1) For |*X*| > *ATmax*, |tan^(-1)(*X*)| is indistinguishable from p/2 in the floating-point format. For single-length, *ATmax* = 1.68*10^(7), and for double-length *ATmax* = 9*10^(15), approximately.

2) This function is numerically very stable, despite the complicated argument reduction. The worst errors occur just above 2-p3, but are no more than 3.2 ulp.

3) It is also very well behaved with respect to errors in the argument, i.e. the error amplification factors are always small.

4) The argument reduction scheme ensures that the identities tan^(-1)(*X*) = p/2 - tan^(-1)(1/*X*), and tan^(-1)(*X*) = p/6 + tan^(-1)((p3**X*-1)/(p3 + *X*)) are preserved.

ATAN2
DATAN2

```
REAL32 FUNCTION ATAN2 (VAL REAL32 X, Y)
REAL64 FUNCTION DATAN2 (VAL REAL64 X, Y)
```

Compute the angular co-ordinate tan^(-1)(*Y*/*X*) (in radians) of a point whose *X* and *Y* co-ordinates are given.

```
Domain:              [-Inf, Inf] x [-Inf, Inf]
Range:               (-Pi, Pi]
Primary Domain:      See note 2.
```

Exceptions

(0, 0) and ([[formula]]**Inf**,[[formula]]**Inf**) give **undefined.NaN**.

Propagated Error

A = *X*(1 [[formula]] *Y*)/(*X*^(2) + *Y*^(2)), *R*= *X*(1 [[formula]] *Y*)/(tan^(-1)(*Y*/*X*)(*X*^(2) + *Y*^(2))) (See note 3)

Generated Error

See note 2.

The Algorithm

1. If *X*, the first argument, is zero, set the result to [[formula]] p /2, according to the sign of *Y*, the second argument.
2. Otherwise set *Rwork*:= **ATAN**(*Y*/*X*). Then if *Y* < 0 set *RESULT* = *Rwork* - *Pi*, otherwise set *RESULT* = *Pi* - *Rwork*.

Notes

1) This two-argument function is designed to perform rectangular-to-polar co-ordinate conversion.

2) See the notes for **ATAN** for the primary domain and estimates of the generated error.

3) The error amplification factors were derived on the assumption that the relative error in *Y* is [[formula]] that in *X*, otherwise there would be separate factors for *X* and *Y*. They are small except near the origin, where the polar co-ordinate system is singular.

SINH
DSINH

```
REAL32 FUNCTION SINH (VAL REAL32 X)
REAL64 FUNCTION DSINH (VAL REAL64 X)
```

Compute sin*h*(*X*).

```
Domain: [-Hmax, Hmax]       = [-89.4, 89.4]S, [-710.5, 710.5]D
Range:                      (-Inf, Inf)
Primary Domain:             (-1.0, 1.0)
```

Exceptions

X < *-Hmax* gives **-Inf**, and *X* > *Hmax* gives **Inf**.

Propagated Error

A = *X*cosh(*X*), *R* = *X*coth(*X*) (See note 3)

Generated Error

	Primary Domain		[1.0, XBig] (See note 2)	
	MRE	RMSRE	MAE	RMSAE
Single Length:	0.91 ulp	0.26 ulp	1.41 ulp	0.34 ulp
Double Length:	0.67 ulp	0.22 ulp	1.31 ulp	0.33 ulp

The Algorithm

1. If |*X*| > *XBig*, set *Rwork*:= **EXP**(|*X*| - ln(2)).
2. If *XBig* w |*X*| w1.0, set temp:= **EXP**(|*X*|), and set *Rwork* = (*temp* - 1/*temp*)/2.
3. Otherwise compute sinh(|*X*|) with a floating-point rational approximation.
4. In all cases, set *RESULT* = [[formula]] *Rwork* according to the sign of *X*.

Notes

1) *Hmax* is the point at which sinh(*X*) becomes too large to be represented in the floating-point format.

2) *XBig* is the point at which *e^(-)^*(|*X* |)becomes insignificant compared with *e^*(|*X* |), (in floating-point). For single-length it is 8.32, and for double-length it is 18.37.

3) This function is quite stable with respect to errors in the argument. Relative error is magnified near zero, but the absolute error is a better measure near the zero of the function and it is diminished there. For large arguments absolute errors are magnified, but since the function is itself large, relative error is a better criterion, and relative errors are not magnified unduly for any argument in the domain, although the output does become less reliable near the ends of the range.

COSH

DCOSH

```
REAL32 FUNCTION COSH (VAL REAL32 X)
REAL64 FUNCTION DCOSH (VAL REAL64 X)
```

Compute cosh(*X*).

```
Domain: [-Hmax, Hmax]       = [-89.4, 89.4]S, [-710.5, 710.5]D
Range:                      [1.0, Inf)
Primary Domain:             [-XBig, XBig]= [-8.32, 8.32]S
  = [-18.37,
                            18.37]D
```

Exceptions

|*X*| > *Hmax* gives **Inf**.

Propagated Error

A = *X* sinh(*X*) , *R* = *X* tanh(*X*) (See note 3)

Generated Error

```
Primary Domain Error:       MRE         RMS
Single Length:              1.24 ulp    0.32 ulp
Double Length:              1.24 ulp    0.33 ulp
```

The Algorithm

1. If |*X*| > *XBig*, set *result*:= **EXP** (|*X*| - ln(2)) .
2. Otherwise, set *temp*:= **EXP** (|*X*|) , and set *result* = (*temp* + 1/*temp*)/2.

Notes

1) *Hmax* is the point at which cosh(*X*) becomes too large to be represented in the floating-point format.

2) *XBig* is the point at which *e^(-)^*(|*X* |)becomes insignificant compared with *e^*(|*X* |)(in floating-point).

3) Errors in the argument are not seriously magnified by this function, although the output does become less reliable near the ends of the range.

TANH
DTANH

```
REAL32 FUNCTION TANH (VAL REAL32 X)
REAL64 FUNCTION DTANH (VAL REAL64 X)
```

Compute tanh(*X*).

```
Domain:             [-Inf, Inf]
Range:              [-1.0, 1.0]
Primary Domain:     [-Log(3)/2, Log(3)/2] = [-0.549, 0.549]
```

Exceptions

None.

Propagated Error

$A = X/\cosh^{(2)}(X)$, $R = X/\sinh(X)\cosh(X)$

Generated Error

Primary Domain Error:	MRE	RMS
Single Length:	0.53 ulp	0.2 ulp
Double Length:	0.53 ulp	0.2 ulp

The Algorithm

1. If $|X| > \ln(3)/2$, set *temp*:= **EXP**$(|X|/2)$. Then set *Rwork* = 1 - 2/(1+*temp*).
2. Otherwise compute *Rwork* = tanh(|*X*|) with a floating-point rational approximation.
3. In both cases, set *RESULT* = [[formula]] *Rwork* according to the sign of *X*.

Notes

1) As a floating-point number, tanh(*X*) becomes indistinguishable from its asymptotic values of [[formula]]1.0 for |*X*|> *HTmax*, where *HTmax* is 8.4 for single-length, and 19.06 for double-length. Thus the output of **TANH** is equal to [[formula]]1.0 for such *X*.

2) This function is very stable and well-behaved, and errors in the argument are always diminished by it.

RAN
DRAN

```
REAL32,INT32 FUNCTION RAN (VAL INT32 X)
REAL64,INT64 FUNCTION DRAN (VAL INT64 X)
```

These produce a pseudo-random sequence of integers, or a corresponding sequence of floating-point numbers between zero and one. X is the seed integer that initiates the sequence.

Domain:	Integers (see note 1)
Range:	[0.0, 1.0] x Integers

Exceptions

None.

The Algorithm

1. Produce the next integer in the sequence: $Nk+1 = (aNk + 1) \bmod M$
2. Treat *Nk+1* as a fixed-point fraction in [0,1), and convert it to floating point.
3. Output the floating point result and the new integer.

Notes

1) This function has two results, the first a real, and the second an integer (both 32 bits for single-length, and 64 bits for double-length). The integer is used as the argument for the next call to **RAN**, i.e. it `carries' the pseudo-random linear congruential sequence *Nk*,

and it should be kept in scope for as long as **RAN** is used. It should be initialized before the first call to **RAN** but not modified thereafter except by the function itself.

2) If the integer parameter is initialized to the same value, the same sequence (both floating-point and integer) will be produced. If a different sequence is required for each run of a program it should be initialized to some `random' value, such as the output of a timer.

3) The integer parameter can be copied to another variable or used in expressions requiring random integers. The topmost bits are the most random. A random integer in the range [0,*L*] can conveniently be produced by taking the remainder by (*L*+1) of the integer parameter shifted right by one bit. If the shift is not done an integer in the range [-*L*,*L*] will be produced.

4) The modulus *M* is 2^(32) for single-length and 2^(64) for double-length, and the multipliers, *a*, have been chosen so that all *M* integers will be produced before the sequence repeats. However several different integers can produce the same floating-point value and so a floating-point output may be repeated, although the *sequence* of such will not be repeated until *M* calls have been made.

5) The floating-point result is uniformly distributed over the output range, and the sequence passes various tests of randomness, such as the `run test', the `maximum of 5 test' and the `spectral test'.

6) The double-length version is slower to execute, but `more random' than the single-length version. If a highly-random sequence of single-length numbers is required, this could be produced by converting the output of **DRAN** to single-length. Conversely if only a relatively crude sequence of double-length numbers is required, **RAN** could be used for higher speed and its output converted to double-length.

1.4.3: IMS T400/T414/T425/T426 elementary function library

To use this library a program header must include the line:

```
#USE "tbmaths.lib"
```

The version of the library described by this section has been written for 32-bit processors without hardware for floating-point arithmetic. Functions from it will give results very close, but not identical to, those produced by the corresponding functions from the single and double length libraries.

This is the version specifically intended to derive maximum performance from the IMS T400, T414, T425, and T426 processors. The single-precision functions make use of the **FMUL** instruction available on 32-bit processors without floating-point hardware. The library is compiled for transputer class **TB**.

The tables and notes at the beginning of section 1.4 apply equally here. However all the functions are contained in one library.

Function definitions

ALOG
DALOG

```
REAL32 FUNCTION ALOG (VAL REAL32 X)
REAL64 FUNCTION DALOG (VAL REAL64 X)
```

These compute: log*e*(*X*)

```
Domain:              (0, XMAX]
Range:               [MinLog, MaxLog] (See note 2)
Primary Domain:      [p2/2, p2) = [0.7071, 1.4142)
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A 51, *R* = log*e*(*X*)

Generated Error

```
Primary Domain Error:      MRE         RMSRE
Single Length:             1.19 ulp    0.36 ulp
Double Length:             2.4 ulp     1.0 ulp
```

The Algorithm

1. Split *X* into its exponent *N* and fraction *F*.
2. Find the natural log of *F* with a fixed-point rational approximation, and convert it into a floating-point number *LnF*.
3. Compute ln(2) * *N* with extended precision and add it to *LnF* to get the result.

Notes

1) The term ln(2) * *N* is much easier to compute (and more accurate) than *LnF*, and it is larger provided *N* is not 0 (i.e. for arguments outside the primary domain). Thus the accuracy of the result improves as the modulus of log(*X*) increases.

2) The minimum value that can be produced, *MinLog*, is the logarithm of the smallest denormalized floating-point number. For single length *MinLog* is -103.28, and for double length it is -744.4. The maximum value *MaxLog* is the logarithm of *XMAX*. For single-length it is 88.72, and for double-length it is 709.78.

3) Since **Inf** is used to represent *all* values greater than *XMAX* its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

ALOG10
DALOG10

```
REAL32 FUNCTION ALOG10 (VAL REAL32 X)
REAL64 FUNCTION DALOG10 (VAL REAL64 X)
```

These compute: log10(*X*)

```
Domain:              (0, XMAX]
Range:               [MinL10, MaxL10] (See note 2)
Primary Domain:      [p2/2, p2) = [0.7071, 1.4142)
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A 5 log10(*e*), *R* = log10(*e*)/log*e*(*X*)

Generated Error

```
Primary Domain Error:      MRE         RMSRE
Single Length:             1.43 ulp    0.39 ulp
Double Length:             2.64 ulp    0.96 ulp
```

The Algorithm

1. Set *temp*:= **ALOG(X)**.
2. If *temp* is a **NaN**, copy it to the output, otherwise set result = log(*e*) * *temp*.

Notes

1) See note 1 for **ALOG**.

2) The minimum value that can be produced, *MinL*10, is the base 10 logarithm of the smallest denormalized floating-point number. For single length *MinL*10 is -44.85, and for double length it is -323.3. The maximum value *MaxL*10 is the base 10 logarithm of *XMAX*. For single length *MaxL*10 is 38.53, and for double-length it is 308.26.

3) Since **Inf** is used to represent *all* values greater than *XMAX* its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

EXP
DEXP

```
REAL32 FUNCTION EXP (VAL REAL32 X)
REAL64 FUNCTION DEXP (VAL REAL64 X)
```

These compute: *e^(X)*

```
Domain:              [-Inf, MaxLog) = [-Inf, 88.03)S,  [-Inf,
                     709.78)D
Range:               [0, Inf) (See note 4)
Primary Domain:      [-Ln2/2, Ln2/2) = [-0.3466, 0.3466)
```

Exceptions

All arguments outside the domain generate an **Inf**.

Propagated Error

A = *Xe^(X)*, *R* = *X*

Generated Error

```
Primary Domain Error:      MRE         RMSRE
```

```
Single Length:              0.51 ulp   0.21 ulp
Double Length:              0.5 ulp    0.21 ulp
```

The Algorithm

1. Set *N* = integer part of *X*/ln(2).
2. Compute the remainder of *X* by ln(2), using extended precision arithmetic.
3. Convert the remainder to fixed-point, compute its exponential using a fixed-point rational function, and convert the result back to floating point.
4. Increase the exponent of the result by *N*. If *N* is sufficiently negative the result must be denormalized.

Notes

1) *MaxLog* is log*e*(*XMAX*).

2) The analytical properties of *e^(x)* make the relative error of the result proportional to the absolute error of the argument. Thus the accuracy of step 2, which prepares the argument for the rational approximation, is crucial to the performance of the subroutine. It is completely accurate when *N* = 0, i.e. in the primary domain, and becomes less accurate as the magnitude of *N* increases. Since *N* can attain larger negative values than positive ones, **EXP** is least accurate for large, negative arguments.

3) For sufficiently negative arguments (below -87.34 for single-length and below -708.4 for double-length) the output is denormalized, and so the floating-point number contains progressively fewer significant digits, which degrades the accuracy. In such cases the error can theoretically be a factor of two.

4) Although the true exponential function is never zero, for large negative arguments the true result becomes too small to be represented as a floating-point number, and **EXP** underflows to zero. This occurs for arguments below -103.9 for single-length, and below -745.2 for double-length.

5) The propagated error is considerably magnified for large positive arguments, but diminished for large negative arguments.

POWER
DPOWER

```
REAL32 FUNCTION POWER (VAL REAL32 X, Y)
REAL32 FUNCTION DPOWER (VAL REAL64 X, Y)
```

These compute: *X^(Y)*

```
Domain:              [0, Inf] x [-Inf, Inf]
Range:               (-Inf, Inf)
Primary Domain:      See note 3.
```

Exceptions

If the first argument is outside its domain, **undefined.NaN** is returned. If the true value of *X^(Y)* exceeds *XMAX*, **Inf** is returned. In certain other cases other **NaN**s are produced: See note 2.

Propagated Error

A = *YX^(Y)*(1 [[formula]] log*e*(*X*)), *R* = *Y*(1 [[formula]] log*e*(*X*)) (See note 4)

Generated Error

Example Range Error:	MRE	RMSRE (See note 3)
Single Length:	1.0 ulp	0.24 ulp
Double Length:	13.2 ulp	1.73 ulp

The Algorithm

Deal with special cases: either argument = 1, 0, +**Inf** or -**Inf** (see note 2). Otherwise:

(a) For single precision:

1. Compute *L* = log2(*X*) in fixed point, where *X* is the first argument.
2. Compute *W* = *Y* x *L* in double precision, where *Y* is the second argument.
3. Compute 2^(*W*) in fixed point and convert to floating-point result.

(b) For double precision:

1. Compute *L* = log2(*X*) in extended precision, where *X* is the first argument.
2. Compute *W* = *Y* x *L* in extended precision, where *Y* is the second argument.
3. Compute *RESULT* = 2^(*W*) in extended precision.

Notes

1) This subroutine implements the mathematical function *x^(y)* to a much greater accuracy than can be attained using the **ALOG** and **EXP** functions, by performing each step in higher precision.

2) Results for special cases are as follows:

First Input (X)	Second Input (Y)	Result
t0	ANY	undefined.NaN
0	v0	undefined.NaN
0	0 <Y<=XMAX	0
0	Inf	unstable.NaN
0 t X t1	Inf	0
0 t X t1	-Inf	Inf
1	-XMAX<=Y<=XMAX	1
1	" Inf	unstable.NaN
1 t X v XMAX	Inf	Inf
1 t X v XMAX	-Inf	0
Inf	1 v Y v Inf	Inf
Inf	-Inf vY v -1	0
Inf	-1 t Y t1	undefined.NaN
otherwise	0	1
otherwise	1	X

3) Performing all the calculations in extended precision makes the double-precision algorithm very complex in detail, and having two arguments makes a primary domain difficult to specify. As an indication of accuracy, the functions were evaluated at 100,000 points logarithmically distributed over (0.1, 10.0), with the exponent linearly distributed over (-35.0, 35.0) (single-length), and (-300.0, 300.0) (double-length), producing the errors given above. The errors are much smaller if the exponent range is reduced.

4) The error amplification factors are calculated on the assumption that the relative error

in *Y* is [[formula]] that in *X*, otherwise there would be separate factors for both *X* and *Y*. It can be seen that the propagated error will be greatly amplified whenever log *e*(*X*) or *Y* is large.

The Algorithm

1. Compute *L* = log2(*X*) in fixed point, where *X* is the first argument.
2. Compute *W* = *Y* x *L* in double precision, where *Y* is the second argument.
3. Compute 2^(*W*) in fixed point and convert to floating-point result.
4. Compute *L* = log2(*X*) in extended precision, where *X* is the first argument.
5. Compute *W* = *Y* x *L* in extended precision, where *Y* is the second argument.
6. Compute *RESULT* = 2^(*W*) in extended precision.

SIN
DSIN

```
REAL32 FUNCTION SIN (VAL REAL32 X)
REAL64 FUNCTION DSIN (VAL REAL64 X)
```

These compute: sine(*X*) (where *X* is in radians)

```
Domain:              [-Smax, Smax] = [-12868.0, 12868.0]S,

                     =   [-2.1*108, 2.1*108]D
Range:               [-1.0, 1.0]
Primary Domain:      [-Pi/2, Pi/2] = [-1.57, 1.57]
```

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except [[formula]] **Inf**, which generates an **undefined.NaN**.

Propagated Error

A = *X* cos(*X*), *R* = *X* cot(*X*)

Generated Error (See note 3)

```
Range:                          Primary Domain              [0, 2Pi]
                              MRE         RMSRE       MAE         RMSAE
Single Length:                0.65 ulp    0.22 ulp    0.74 ulp    0.18 ulp
Double Length:                0.56 ulp    0.21 ulp    0.64 ulp    0.16 ulp
```

The Algorithm

1. Set *N* = integer part of |*X*|/*Pi*.
2. Compute the remainder of |*X*| by *Pi*, using extended precision arithmetic.
3. Convert the remainder to fixed-point, compute its sine using a fixed-point rational function, and convert the result back to floating point.
4. Adjust the sign of the result according to the sign of the argument and the evenness of *N*.

Notes

1) For arguments outside the primary domain the accuracy of the result depends crucially on step 2. The extended precision corresponds to *K* extra bits in the representation of p (*K* = 8 for single-length and 12 for double-length). If the argument agrees with an integer multiple of p to more than *K* bits there is a loss of significant bits

in the remainder, equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The extra precision of step 2 is lost if *N* becomes too large, and the cut-off *Smax* is chosen to prevent this. In any case for large arguments the `granularity' of floating-point numbers becomes a significant factor. For arguments larger than *Smax* a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

3) The propagated error has a complex behavior. The propagated relative error becomes large near each zero of the function (outside the primary range), but the propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

4) Since only the remainder of *X* by *Pi* is used in step 3, the symmetry sin(*x*+ *n*p) = [[formula]] sin(*x*) is preserved, although there is a complication due to differing precision representations of p.

5) The output range is not exceeded. Thus the output of **SIN** is always a valid argument for **ASIN**.

COS
DCOS

```
REAL32 FUNCTION COS (VAL REAL32 X)
REAL64 FUNCTION DCOS (VAL REAL64 X)
```

These compute: cosine (*X*) (where *X* is in radians)

```
Domain:              [-Smax, Smax] = [-12868.0, 12868.0]S,

                     =  [-2.1*108, 2.1*108]D
Range:               [-1.0, 1.0]
Primary Domain:      See note 1.
```

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except [[formula]]**Inf**, which generates an **undefined.NaN**.

Propagated Error

A = -*X* sin(*X*), *R* = -*X* tan(*X*) (See note 4)

Generated Error

Range:	[0,Pi/4)		[0, 2Pi]	
	MRE	RMSRE	MAE	RMSAE
Single Length:	1.0 ulp	0.28 ulp	0.81 ulp	0.17 ulp
Double Length:	0.93 ulp	0.26 ulp	0.76 ulp	0.18 ulp

The Algorithm

1. Set *N* = integer part of (|*X*|+*Pi*/2)/*Pi*.
2. Compute the remainder of (|*X*|+*Pi*/2) by *Pi*, using extended precision arithmetic.

3. Compute the remainder to fixed-point, compute its sine using a fixed-point rational function, and convert the result back to floating point.
4. Adjust the sign of the result according to the evenness of *N*.

Notes

1) Inspection of the algorithm shows that argument reduction always occurs, thus there is no `primary domain' for cos. So for all arguments the accuracy of the result depends crucially on step 2. The extended precision corresponds to *K* extra bits in the representation of p (*K* = 8 for single-length and 12 for double length). If the argument agrees with an odd integer multiple of p/2 to more than *K* bits there is a loss of significant bits in the remainder, equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The extra precision of step 2 is lost if *N* becomes too large, and the cut-off *Smax* is chosen to prevent this. In any case for large arguments the `granularity' of floating-point numbers becomes a significant factor. For arguments larger than *Smax* a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

3) For small arguments the errors are not evenly distributed. As the argument becomes smaller there is an increasing bias towards negative errors (which is to be expected from the form of the Taylor series). For the single-length version and *X* in [-0.1, 0.1], 62% of the errors are negative, whilst for *X* in [-0.01, 0.01], 70% of them are.

4) The propagated error has a complex behavior. The propagated relative error becomes large near each zero of the function, but the propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

5) Since only the remainder of (|*X*|+*Pi*/2) by *Pi* is used in step 3, the symmetry cos(*x*+ *n*p) = [[formula]] cos(*x*) is preserved. Moreover, since the same rational approximation is used as in **SIN**, the relation cos(*x*) =sin(*x*+ p/2) is also preserved. However, in each case there is a complication due to the different precision representations of p.

6) The output range is not exceeded. Thus the output of cos is always a valid argument for **ACOS**.

TAN
DTAN

```
REAL32 FUNCTION TAN (VAL REAL32 X)
REAL64 FUNCTION DTAN (VAL REAL64 X)
```

These compute: tan(*X*) (where *X* is in radians)

```
Domain:              [-Tmax, Tmax]      = [-6434.0, 6434.0]S
                                        = [-1.05*108, 1.05*108]D
Range:               (-Inf, Inf)
Primary Domain:      [-Pi/4, Pi/4]      = [-0.785, 0.785]
```

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except [[formula]] **Inf**, which generate an **undefined.NaN**. Odd integer multiples of p/2 may produce

unstable.NaN.

Propagated Error

A = *X*(1+ tan^(2)(*X*)), *R* = *X*(1+ tan^(2)(*X*))/tan(*X*) (See note 4)

Generated Error

```
Primary Domain Error:      MRE         RMSRE
Single Length:             3.5 ulp     0.23 ulp
Double Length:             0.69 ulp    0.23 ulp
```

The Algorithm

1. Set *N* = integer part of *X*/(*Pi*/2).
2. Compute the remainder of *X* by *Pi*/2, using extended precision arithmetic.
3. Convert the remainder to fixed-point, compute its tangent using a fixed-point rational function, and convert the result back to floating point.
4. If *N* is odd, take the reciprocal.
5. Set the sign of the result according to the sign of the argument.

Notes

1) *R* is large whenever *X* is near to an integer multiple of p/2, and so tan is very sensitive to small errors near its zeros and singularities. Thus for arguments outside the primary domain the accuracy of the result depends crucially on step 2. The extended precision corresponds to *K* extra bits in the representation of p/2 (*K* = 8 for single-length and 12 for double-length). If the argument agrees with an integer multiple of p/2 to more than *K* bits there is a loss of significant bits in the remainder, approximately equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The extra precision of step 2 is lost if *N* becomes too large, and the cut-off *Tmax* is chosen to prevent this. In any case for large arguments the `granularity' of floating-point numbers becomes a significant factor. For arguments larger than *Tmax* a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

3) Step 3 of the algorithm has been slightly modified in the double-precision version from that given in Cody & Waite to avoid fixed-point underflow in the polynomial evaluation for small arguments.

4) Tan is quite badly behaved with respect to errors in the argument. Near its zeros outside the primary domain the relative error is greatly magnified, though the absolute error is only proportional to the size of the argument. In effect, the error is seriously amplified in an interval about each irrational zero, whose width increases roughly in proportion to the size of the argument. Near its singularities both absolute and relative errors become large, so any large output from this function is liable to be seriously contaminated with error, and the larger the argument, the smaller the maximum output which can be trusted. If step 4 of the algorithm requires division by zero, an **unstable.NaN** is produced instead.

5) Since only the remainder of *X* by *Pi*/2 is used in step 3, the symmetry tan(*x*+ *n*p) = tan(*x*) is preserved, although there is a complication due to the differing precision representations of p. Moreover, by step 4 the symmetry tan(*x*) = 1/ tan(p/2 - *x*) is also preserved.

ASIN
DASIN

```
REAL32 FUNCTION ASIN (VAL REAL32 X)
REAL64 FUNCTION DASIN (VAL REAL64 X)
```

These compute: sin^(-1)(*X*) (in radians)

```
Domain:              [-1.0, 1.0]
Range:               [-Pi/2, Pi/2]
Primary Domain:      [-0.5, 0.5]
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A = *X*/p1 - *X*^(2), *R* = *X*/(sin^(-1)(*X*) p1 - *X*^(2))

Generated Error

```
                              Primary Domain            [-1.0, 1.0]
                            MRE         RMSRE       MAE         RMSAE
Single Length:              0.53 ulp    0.21 ulp    1.35 ulp    0.33 ulp
Double Length:              2.8 ulp     1.4 ulp     2.34 ulp    0.64 ulp
```

The Algorithm

1. If |*X*|> 0.5, set *Xwork*:= **SQRT**(1 - |*X*|)/2).
   Compute *Rwork* = arcsine(-2 * *Xwork*) with a floating-point rational approximation, and set the result = *Rwork* + *Pi*/2.
2. Otherwise compute the result directly using the rational approximation.
3. In either case set the sign of the result according to the sign of the argument.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error: however since both domain and range are bounded the *absolute* error in the result cannot be large.

2) By step 1, the identity sin^(-1)(*x*) = p/2 - 2 sin^(-1)(p(1-*x*)/2)) is preserved.

ACOS
DACOS

```
REAL32 FUNCTION ACOS (VAL REAL32 X)
REAL64 FUNCTION DACOS (VAL REAL64 X)
```

These compute: cosine^(-1)(*X*) (in radians)

```
Domain:              [-1.0, 1.0]
Range:               [0, Pi]
Primary Domain:      [-0.5, 0.5]
```

Exceptions

All arguments outside the domain generate an **undefined.NaN**.

Propagated Error

A = -*X*/p1 - *X*^(2), *R* = -*X*/(sin^(-1)(*X*) p1 - *X*^(2))

Generated Error

```
                              Primary Domain            [-1.0, 1.0]
                            MRE         RMSRE       MAE         RMSAE
Single Length:              1.1 ulp     0.38 ulp    2.4 ulp     0.61 ulp
Double Length:              1.3 ulp     0.34 ulp    2.9 ulp     0.78 ulp
```

The Algorithm

1. If |*X*| > 0.5, set *Xwork*:= **SQRT**(1- |*X*|)/2) . Compute *Rwork* = arcsine (2 * *Xwork*) with a floating-point rational approximation. If the argument was positive, this is the result, otherwise set the result = *Pi* - *Rwork*.
2. Otherwise compute *Rwork* directly using the rational approximation. If the argument was positive, set result = *Pi*/2 - *Rwork*, otherwise result = *Pi*/2 + *Rwork*.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error, although this interval is larger near 1 than near -1, since the function goes to zero with an infinite derivative there. However since both the domain and range are bounded the *absolute* error in the result cannot be large.

2) Since the rational approximation is the same as that in **ASIN**, the relation cos^(-1)(*x*) = p/2 - sin^(-1)(*x*)is preserved.

ATAN
DATAN

```
REAL32 FUNCTION ATAN (VAL REAL32 X)
REAL64 FUNCTION DATAN (VAL REAL64 X)
```

These compute: tan^(-1)(*X*) (in radians)

```
Domain:             [-Inf, Inf]
Range:              [-Pi/2, Pi/2]
Primary Domain:     [-z, z],   z = 2 - p3 = 0.2679
```

Exceptions

None.

Propagated Error

A = *X*/(1 + *X*^(2)), *R* = *X*/(tan^(-1)(*X*)(1 + *X*^(2)))

Generated Error

```
Primary Domain Error:       MRE         RMSRE
Single Length:              0.53 ulp    0.21 ulp
```

```
Double Length:              1.27 ulp   0.52 ulp
```

The Algorithm

1. If |*X*| > 1.0, set *Xwork* = 1/|*X*|, otherwise *Xwork* = |*X*|.
2. If *Xwork* > 2-p3, set *F* = (*Xwork**p3 -1)/(*Xwork* +p3), otherwise *F* = *Xwork*.
3. Compute *Rwork* = arctan(*F*) with a floating-point rational approximation.
4. If *Xwork* was reduced in (2), set *R* = *Pi*/6 + *Rwork*, otherwise *R* = *Rwork*.
5. If *X* was reduced in (1), set *RESULT* = *Pi*/2 - *R*, otherwise *RESULT* = *R*.
6. Set the sign of the *RESULT* according to the sign of the argument.

Notes

1) For |*X*| > *ATmax*, |tan^(-1)(*X*)| is indistinguishable from p/2 in the floating-point format. For single-length, *ATmax* = 1.68*10^(7), and for double-length *ATmax* = 9*10^(15), approximately.

2) This function is numerically very stable, despite the complicated argument reduction. The worst errors occur just above 2-p3, but are no more than 1.8 ulp. 3) It is also very well behaved with respect to errors in the argument, i.e. the error amplification factors are always small.

4) The argument reduction scheme ensures that the identities tan^(-1)(*X*) = p/2 - tan^(-1)(1/*X*), and tan^(-1)(*X*) = p/6 + tan^(-1)((p3**X*-1)/(p3 + *X*)) are preserved.

ATAN2
DATAN2

```
REAL32 FUNCTION ATAN2 (VAL REAL32 X, Y)
REAL64 FUNCTION DATAN2 (VAL REAL64 X, Y)
```

These compute the angular co-ordinate tan^(-1)(*Y*/*X*) (in radians) of a point whose *X* and *Y* co-ordinates are given.

```
Domain:              [-Inf, Inf] x [-Inf, Inf]
Range:               (-Pi, Pi]
Primary Domain:      See note 2.
```

Exceptions

(0, 0) and ([[formula]]**Inf**,[[formula]]**Inf**) give **undefined.NaN**.

Propagated Error

A = *X*(1 [[formula]] *Y*)/(*X*^(2) + *Y*^(2)), *R* = *X*(1 [[formula]] *Y*)/(tan^(-1)(*Y*/*X*)(*X*^(2) + *Y*^(2))) (See note 3)

Generated Error

See note 2.

The Algorithm

1. If *X*, the first argument, is zero, set the result to [[formula]] p/2, according to the sign of *Y*, the second argument.

2. Otherwise set *Rwork*:= **ATAN** (*Y*/*X*) . Then if *Y* < 0 set *RESULT* = *Rwork* - *Pi*, otherwise set *RESULT* = *Pi* - *Rwork*.

Notes

1) This two-argument function is designed to perform rectangular-to-polar co-ordinate conversion.

2) See the notes for **ATAN** for the primary domain and estimates of the generated error.

3) The error amplification factors were derived on the assumption that the relative error in *Y* is [[formula]] that in *X*, otherwise there would be separate factors for *X* and *Y*. They are small except near the origin, where the polar co-ordinate system is singular.

SINH
DSINH

```
REAL32 FUNCTION SINH (VAL REAL32 X)
REAL64 FUNCTION DSINH (VAL REAL64 X)
```

These compute: sinh(*X*)

```
Domain:[-Hmax, Hmax]        = [-89.4, 89.4]S, [-710.5, 710.5]D
Range:                      (-Inf, Inf)
Primary Domain:             (-1.0, 1.0)
```

Exceptions

X < -*Hmax* gives -**Inf**, and *X* > *Hmax* gives **Inf**.

Propagated Error

A = *X* cosh(*X*), *R* = *X* coth(*X*) (See note 3)

Generated Error

	Primary Domain		[1.0, XBig] (See note 2)	
	MRE	RMSRE	MAE	RMSAE
Single Length:	0.89 ulp	0.3 ulp	0.98 ulp	0.31 ulp
Double Length:	1.3 ulp	0.51 ulp	1.0 ulp	0.3 ulp

The Algorithm

1. If |*X*| > *XBig*, set *Rwork*:= **EXP** (|*X*| - ln(2)) .
2. If *XBig* w |*X*| w 1.0, set *temp*:= **EXP** (|*X*|) , and set *Rwork* = (*temp* - 1/*temp*)/2.
3. Otherwise compute *Rwork* = sinh(|*X*|) with a fixed-point rational approximation.
4. In all cases, set *RESULT* = [[formula]] *Rwork* according to the sign of *X*.

Notes

1) *Hmax* is the point at which sinh(*X*) becomes too large to be represented in the floating-point format.

2) *XBig* is the point at which *e^(-)^*(|*X* |)becomes insignificant compared with *e^*(|*X* |), (in floating-point). For single-length it is 8.32, and for double-length it is 18.37.

3) This function is quite stable with respect to errors in the argument. Relative error is magnified near zero, but the absolute error is a better measure near the zero of the function and it is diminished there. For large arguments absolute errors are magnified, but since the function is itself large, relative error is a better criterion, and relative errors are not magnified unduly for any argument in the domain, although the output does become less reliable near the ends of the range.

COSH
DCOSH

```
REAL32 FUNCTION COSH (VAL REAL32 X)
REAL64 FUNCTION DCOSH (VAL REAL64 X)
```

These compute: cosh(*X*)

```
Domain: [-Hmax, Hmax]             = [-89.4, 89.4]S, [-710.5, 710.5]D
Range:  [1.0, Inf)
PrimaryDomain: [-XBig, XBig]      = [-8.32, 8.32]S
= [-18.37, 18.37]D
```

Exceptions

|*X*| > *Hmax* gives **Inf**.

Propagated Error

A = *X* sinh(*X*) , *R* = *X* tanh(*X*) (See note 3)

Generated Error

Primary Domain Error:	MRE	RMS
Single Length:	0.99 ulp	0.3 ulp
Double Length:	1.23 ulp	0.3 ulp

The Algorithm

1. If |*X*| > *XBig*, set *result*:= **EXP** (|*X*| - ln(2)) .
2. Otherwise, set *temp*:= **EXP** (|*X*|) , and set *result* = (*temp* + 1/*temp*)/2.

Notes

1) *Hmax* is the point at which cosh(*X*) becomes too large to be represented in the floating-point format.

2) *XBig* is the point at which *e^(-)^*(|*X* |)becomes insignificant compared with *e^*(|*X* |) (in floating-point).

3) Errors in the argument are not seriously magnified by this function, although the output does become less reliable near the ends of the range.

TANH
DTANH

```
REAL32 FUNCTION TANH (VAL REAL32 X)
```

```
REAL64 FUNCTION DTANH (VAL REAL64 X)
```

These compute: tanh(*X*)

```
Domain:              [-Inf, Inf]
Range:               [-1.0, 1.0]
Primary Domain:      [-Log(3)/2, Log(3)/2] = [-0.549, 0.549]
```

Exceptions

None.

Propagated Error

A = *X*/cosh^(2)(*X*), *R* = *X* / sinh (*X*) cosh(*X*)

Generated Error

```
Primary Domain Error:      MRE         RMS
Single Length:             0.52 ulp    0.2 ulp
Double Length:             4.6 ulp     2.6 ulp
```

The Algorithm

1. If |*X*| > ln(3)/2, set *temp*:= **EXP**(|*X*|/2). Then set *Rwork* = 1 - 2/(1+*temp*).
2. Otherwise compute *Rwork* = tanh(|*X*|) with a floating-point rational approximation.
3. In both cases, set *RESULT* = [[formula]] *Rwork* according to the sign of *X*.

Notes

1) As a floating-point number, tanh(*X*) becomes indistinguishable from its asymptotic values of [[formula]]1.0 for |*X*| > *HTmax*, where *HTmax* is 8.4 for single-length, and 19.06 for double-length. Thus the output of **TANH** is equal to [[formula]]1.0 for such *X*.

2) This function is very stable and well-behaved, and errors in the argument are always diminished by it.

RAN
DRAN

```
REAL32,INT32 FUNCTION RAN (VAL INT32 X)
REAL64,INT64 FUNCTION DRAN (VAL INT64 X)
```

These produce a pseudo-random sequence of integers, and a corresponding sequence of floating-point numbers between zero and one.

```
Domain:              Integers (see note 1)
Range:               [0.0, 1.0] x Integers
```

Exceptions

None.

The Algorithm

1. Produce the next integer in the sequence: *Nk+1* = (*aNk* + 1)*mod M*

2. Treat *Nk+1* as a fixed-point fraction in [0,1), and convert it to floating point.
3. Output the floating point result and the new integer.

Notes

1) This function has two results, the first a real, and the second an integer (both 32 bits for single-length, and 64 bits for double-length). The integer is used as the argument for the next call to **RAN**, i.e. it `carries' the pseudo-random linear congruential sequence *Nk*, and it should be kept in scope for as long as **RAN** is used. It should be initialized before the first call to **RAN** but not modified thereafter except by the function itself.

2) If the integer parameter is initialized to the same value, the same sequence (both floating-point and integer) will be produced. If a different sequence is required for each run of a program it should be initialized to some `random' value, such as the output of a timer.

3) The integer parameter can be copied to another variable or used in expressions requiring random integers. The topmost bits are the most random. A random integer in the range [0,*L*] can conveniently be produced by taking the remainder by (*L*+1) of the integer parameter shifted right by one bit. If the shift is not done an integer in the range [-*L*,*L*] will be produced.

4) The modulus *M* is 2^(32) for single-length and 2^(64) for double-length, and the multipliers, *a*, have been chosen so that all *M* integers will be produced before the sequence repeats. However several different integers can produce the same floating-point value and so a floating-point output may be repeated, although the *sequence* of such will not be repeated until *M* calls have been made.

5) The floating-point result is uniformly distributed over the output range, and the sequence passes various tests of randomness, such as the `run test', the `maximum of 5 test' and the `spectral test'.

6) The double-length version is slower to execute, but `more random' than the single-length version. If a highly-random sequence of single-length numbers is required, this could be produced by converting the output of **DRAN** to single-length. Conversely if only a relatively crude sequence of double-length numbers is required, **RAN** could be used for higher speed and its output converted to double-length.